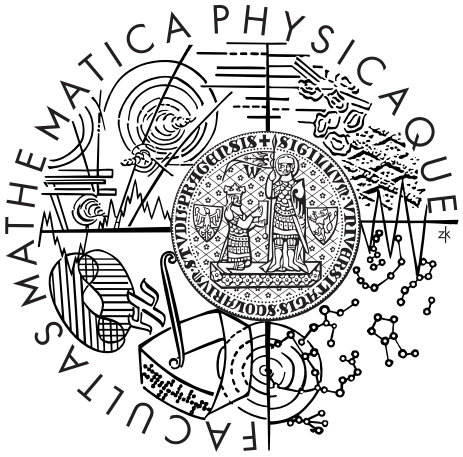
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jan Šebetovský

Support for C++ in GMC

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Jan Kofroň, Ph.D.

Study programme: Computer science

Specialization: Software systems

Prague 2013

I would like to thank to my supervisor Jan Kofroň and to Pavel Jančík for valuable advices and to my friends for support during work on this thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date 30$^{\text{th}}$ July, 2013

Název práce: Support for C++ in GMC
Autor: Jan Šebetovský
Katedra: Katedra distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: RNDr. Jan Kofroň, Ph.D., KDSS

Abstrakt: Software je používán na stále více místech našeho života a tak je stále důležitější jeho správnost. Proto je dobré přistoupit k jeho formální verifikaci. V současnosti neexistuje mnoho nástrojů pro verifikaci kódu v jazyce C++ a většina z nich neumí verifikovat všechny potřebné vlastnosti. Proto jsme se rozhodli rozšířit program GMC, který už uměl kontrolovat programy v jazyce C, o podporu jazyka C++. Kvůli značné rozsáhlosti jazyka C++ bylo cílem této práce implementovat jen základní vlastnosti jazyka (dědičnost, konstruktory, destruktory, virtuální metody a výjimky). Podpora všech těchto vlastností byla implementována až na výjimky, které jsou implementovány jen částečně.

Klíčová slova: formální verifikace, C++, GMC

Title: Support for C++ in GMC
Author: Jan Šebetovský
Department: Department of Distributed and Dependable Systems
Supervisor: RNDr. Jan Kofroň, Ph.D., DDDS

Abstract: Software is used in more and more aspects of our lives, so its correctness is more and more important. Its verification is thus a good idea. Now there are not many tools for verification of programs in the C++ language and most of them cannot verify all required properties. Because of this we decided to extend GMC, which was already able to verify C code, with support of the C++ language. However the C++ language is very vast, so the goal of this work is implementation of only the basic language features (inheritance, constructors, destructors, virtual methods and exceptions). The support of all those features have been implemented except for exceptions, which are implemented only partially.

Keywords: formal verification, C++, GMC

Titolo: Subteno por C++ en GMC
Aŭtoro: Jan Šebetovský
Katedro: Katedro de disaj kaj fideblaj sistemoj
Gvidanto: RNDr. Jan Kofroň, Ph.D., Katedro de disaj kaj fideblaj sistemoj

Abstraktaĵo: Programaro estas uzata por pli kaj pli aferoj en nia vivo, do estas pli kaj pli grava ĝia ĝusteco. Pro tio estas bone uzi formalan kontroladon. Nuntempe ne ekzistas multe da iloj por kontroli programoj en la lingvo C++ kaj multe el ili ne povas kontroli ĉiujn bezonatajn ecojn. Pro tio ni decidis etendigi la programon GMC, kiu jam povis kontroli programojn en la lingvo C, per subteno por la lingvo C++. Pro la signifa vasto de la lingvo C++ ni celis per ĉi tiu laboro realigi nur la bazajn ecojn de la lingvo (heredadon, konstruilojn, malkonstruilojn, virtualajn metodojn kaj esceptojn). La subteno de ĉiuj ĉi tiuj ecoj estas realigita kun escepto de esceptoj, kiuj estas realigitaj nur parte.

Ŝlosilvortoj: formala kontrolado, C++, GMC

Contents

1. Introduction

There are a lot of programs created these days in the world. Some of them are interactive (like editors), so user usually see the result of program's work before its usage in the rest of the world (you can see the image before you print it and send it to someone). In the case of malfunction, user can handle the situation. But there are also programs, which create the result directly visible to the rest of the world without checking by the user. It could be some server, application which handles money, security system or program for hardware (medical devices, cars and satellites). In those cases user usually cannot handle errors in the program before they cause some damage.

In some cases it is thus really useful for user to be sure that a program works with no error. This can be shown by means of formal verification, which is a method, which considers all states and checks if all given requirements are satisfied. A problem is that the state space of most programs is too big to explore it state by state (state explosion). First it is too big to save all states into memory and second it could take too long to explore it entirely.

Different verification systems handle those problems differently. One approach is explicit state model checking, which is used by GMC. It means that GMC really explores all states of the program, but it does not save them, but only calculates the hash to know[1] if that state has been already explored.

1.1 Goals of this work

The aim of the GMC project is to create a prototype of explicit state verificator, which will be able to check multithreaded programs in all languages for which GCC has a front-end.

The goal of this work is to implement support for the basic C++ features, using two preceding works, which resulted in GMC with the support of the C language. The differences between C and C++ code are obvious, but differences in the generated GIMPLE (intermediate code representation used inside GCC [10]) are much more important for this work. The first difference is that GIMPLE contains, in addition to functions, also methods, which could also be virtual and thus called in a special way. The GIMPLE from C++ can also contain constructors and destructors of objects, which look like very special methods. And finally the fact that C++ code contains exceptions results in usage of special GIMPLE constructs and calling special built-in functions. All those differences had to be covered by modifications to GMC, which are discussed in the following chapters.

In this work should be thus into GMC implemented support for: inheritance, constructors, destructors, virtual methods and exceptions.

1.2 Structure of this work

The most important part of this work consists of chapters which describe our solution, including issues we faced. Each of those chapters consists of three parts.

[1] There is a negligible probability of error due to hash collisions.

In the first part there is a description of the problem and a general description of the original situation. The solution of the problem and the implementation are described in the other parts.

2. Already done work

This work is based on two previous works. The goal of one of them [1] was to create a memory module for the model checker. The memory module is not described in detail here, because it is rarely used by the new code created as a part of this work. In order to make the memory module easily testable, also a simple gimple iterator and gimple interpreter have been created.

Those two modules have been widely extended in another work [2] resulting to a working model checker of C code. The modules are identically connected in both those works; this connection remains the same also in the model checker of the C++ code, whose creation is the goal of this work. You can see this interconnection in Figure 2.1.

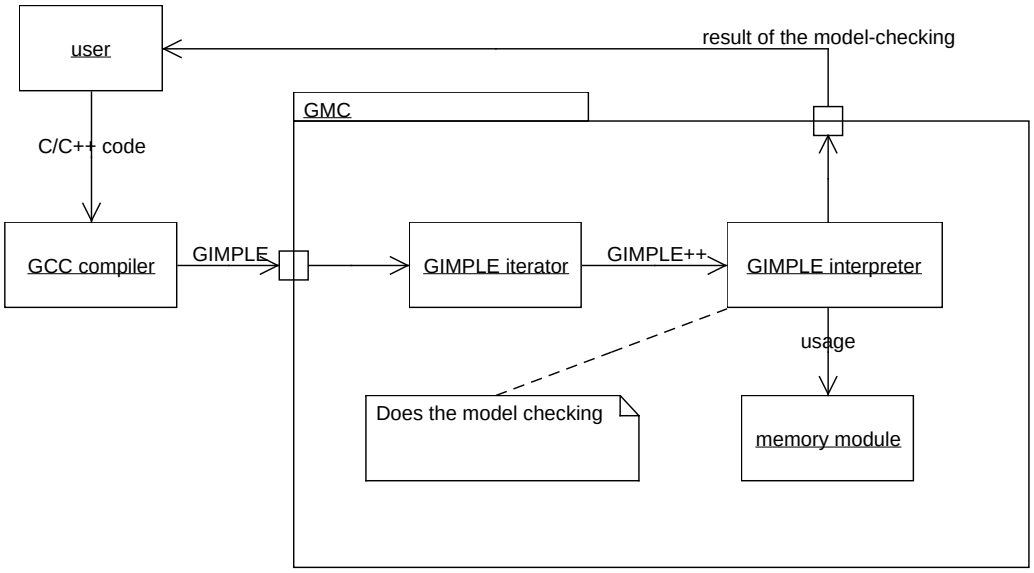


Figure 2.1: GMC structure.

The C-code model checker, on which this work is based, is divided into three parts. The smallest one is compiled into the GCC and uses the second part linked to the GCC as a library. This library contains the main part of the gimple iterator, which converts GCC code representation (*GIMPLE*) into the GMC code representation (*GIMPLE++*). This code representation is saved to a file and loaded by the third part – the model checker, which runs as a separate process.

2.1 Implementation details

In this section we mention some implementation details which are useful to understand how GMC exactly works and what has been changed in its sources in order to implement the support for C++.

The whole first part of GMC (which is mentioned at the end of the previous section) is written in the file `gimplexx-bridge.c`, where the function

`gb_process_cfun` is called from GCC for each function in the checked code. The main task of this function is to convert a function from GIMPLE to GIMPLE++. This is done by going through all blocks (and other parts) of the passed function and calling the function `basic_block_to_git_block` (or other appropriate function with name of pattern `*_to_git_*`) to convert them to the GIMPLE++ representation. This function similarly goes through all statements and converts them into its GIMPLE++ representation. Sometimes those functions call a function of the pattern `create_git_*_from_tree`, but even in this case the creation of the GIMPLE++ object is done in a function of the pattern `createSomething` (where "Something" is substituted by the name of the created object). Those functions are called from `gimplexx-bridge.c`, but they are defined in the `ConstructionHelpers.c` file and represent the transition from the first part to the second part of the model checker and also the transition from the C code to the C++ code.

After this conversion the structures are serialized into the `gimplexx.ser` file in the function `lto_main` in the file `lto.c`[1]. By this step the work of the second part of GMC is finished. The `gimplexx.ser` file is then loaded by the `ModelChecker` process, which deserializes GIMPLE++ objects from it. It also initializatizes global structures for declarations, constants, etc. After deserialization the model checking (or interpretation) is run. The main loop of the model checking algorithm is in the file `ModelChecker.cpp` in the function `doModelCheck_`, where threads of the model-checked program are scheduled and steps are executed using the class `GimpleStatementInterpreter`. The results of the model-checking are written to the `ModelChecker_stderr` file.

[1] LTO = link time optimization

3. Functions and methods

Functions in the C language share the same global context. Except for global variables, a function can have explicit parameters and static variables with which it can work. In C++ there are in addition methods, which can be seen as functions defined for a class. A method has one extra implicit parameter pointing to the object on which it is called (*this*).

The implementation of conversion and interpretation of methods in GMC was easy, because methods are a kind of functions and in GIMPLE they differ only by labelling. Implementation of virtual method calling was a bit more complicated.

3.1 Virtual methods

A virtual method is a special method which is overridable in a derived class, so in GIMPLE it cannot be called directly by its address (which is the case of standard methods), but the virtual method table (VMT) is used instead.

VMT is created by GCC for each class which contains a virtual method. For each virtual method, the table contains a pointer to the corresponding implementation. Each created object then contains a variable pointing to the appropriate VMT. This variable is initialized by the code, which is added to the beginning of each constructor during creation of the GIMPLE code. When a new class which contains additional virtual methods is derived from this class, each object of the derived class will contain two pointer variables (one for the base class and one for the derived one). The variable for the derived class will point to the beginning of the VMT and variable for the base class will point to the appropriate part of VMT which contains pointers to the methods derived from the base class. Those methods can of course be specialized for the derived class.

In GIMPLE calling a virtual method looks like this:

1. From the object on which the method is called, the pointer to the VMT is acquired.

2. The method is called using the ordinary call statement, but instead of the pointer to the method, a structure, which contains a pointer to the VMT and an index of the called method, is passed as the first parameter.

In the previous version of the GMC implementation of function calling using the mentioned structure is missing. There is also a problem with initialization of the global variables. It occurs (even in C) if global variables are initialized by an address expression. And exactly those constructs are used during the initialization of VMT. So we had to fix it. VMTs are on the other hand in GIMPLE represented by C-code structures, thus there is no need of further implementation.

3.2 Implementation

In GIMPLE the function structures are marked by the constant `FUNCTION_TYPE` and the method structures are marked by the constant

METHOD_TYPE. The previous version of GMC recognizes only FUNCTION_TYPE, so there is no support for the method declaration when it creates GIMPLE++ from the GIMPLE.

We could solve this problem by simple addition of code which would recognize the METHOD_TYPE constant and create the same structures as for functions. However, for better extendibility, we created base classes (which we named using the word "callable") for all structures used to store functions. From those base classes we derived classes for functions and methods. Now the majority of functionality for functions and methods is implemented in the common base classes.

For the case of virtual methods we first fixed the code to create global initializers. The problem was that GMC did not know how to handle the operation passed as an operand. The situation when an operation is passed as an operand is determined using a new function is_operation.

After fixing virtual method table initialization, we added an implementation of OBJ_TYPE_REF which is a macro identifying the structure containing the pointer to a VMT and the index of the called function. So when an operand is of this type, we extract the reference to the function by the macro OBJ_TYPE_REF_EXPR. After this operation, calling a virtual method is the same as calling a non-virtual method.

4. Constructors and destructors

The system of constructors and destructors is a little bit complicated in C++ because of virtual inheritance. But it is completely the same for constructors and destructors so we describe the system only for the former ones.

First of all we explain usage of constructors in C++ from the programmer's point of view. When there is only non-virtual inheritance, the situation is simple:

Every class has to have a constructor (if no constructor is defined by the programmer, the default empty nonparametric constructor is assumed). If a class is derived from a base class, the derived class has to call a constructor of the base class. If the programmer does not call a constructor of the base class, the call of nonparametric constructor is assumed. If there is no such one, it is an erroneous state. If the base class is also derived, the call of base base class constructor is fully handled by the base class. In general the constructor of the base class is called by the directly derived class constructor.

If a class is virtually derived from a base class, the situation is similar, but if also the derived class has a child in the inheritance hierarchy, the situation is very different: This child has to call constructors of both base classes. In general when there is a chain of inheritance and there is a virtual inheritance, constructors of all base classes which are inherited virtually are called by the most specific class in this chain.

The mentioned difference is here because of the diamond problem [3]. When all derived classes are derived using non-virtual inheritance, it results in two instances of the most general class in the instance of the most specific class (see the Figure 4.1). So it is clear that constructor of one instance of `A` is called by constructor of `Ba` and constructor of the other instance of `A` is called by constructor of `Bb`.

On the other hand if classes `Ba` and `Bb` are derived using virtual inheritance (see the Figure 4.2, there is only one instance of `A` in `C`. In this case would be undecidable, which constructor should be called, when `Ba` and `Bb` would use different constructors of `A`. To solve this situation, the constructor of the most specific class (`C` in described example) decides which constructor of `A` will be called.

Now let's consider the situation, when we want to instantiate the class `Ba` from the Figure 4.2. In that case we want the constructor `Ba` call the constructor of `A`. But in the previous paragraph that constructor was selected by the class `C`.

So two different constructors have to be generated by GCC from one constructor written by a programmer, when the described situation occurs. The constructor which does not call constructors of virtually inherited classes is called base constructor. The constructor which does call constructors of virtually inherited classes is called complete constructor.

It means that, when there is no virtual inheritance, each constructor is called base constructor. On the other hand, whenever the class is instantiated by the programmer, the complete constructor is called (because also virtually inherited parts have to be constructed if there are some). So when there is no virtual inheritance, the name used in the call instruction does not match the name of

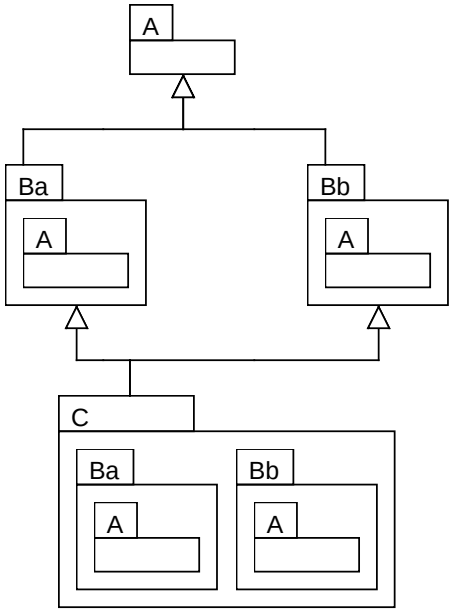


Figure 4.1: Non-virtual diamond. Diamond problem example with no virtual inheritance.

the constructor which is used in declaration. This mismatch is not resolved in GIMPLE, which is used by GMC, so GMC has to resolve it on its own.

This looking for the appropriate constructors is even more complicated because of the fact that constructors can be overloaded, so also parameter types have to be considered.

4.1 Implementation

When there is a constructor or destructor defined in the C++ code, it has one of the following names in GIMPLE:

- "___comp_ctor "[1] (in the case of a complete constructor),
- "___comp_dtor "[1] (in the case of a complete destructor),
- "___base_ctor "[1] (in the case of a base constructor) or
- "___base_dtor "[1] (in the case of a base destructor).

Constructors are always called by the name "___comp_ctor " and destructors are always called by the name "___comp_dtor ". So whenever one of those two names is used to call a function, GMC has to check if there is a suitable (according to parameters) complete constructor, if there is none, it has to call the base constructor.

However, the first thing which GMC has to do is a conversion of definitions of those methods from GIMPLE to GIMPLE++. There are already data structures

[1]Note that there is a space at the end of the name

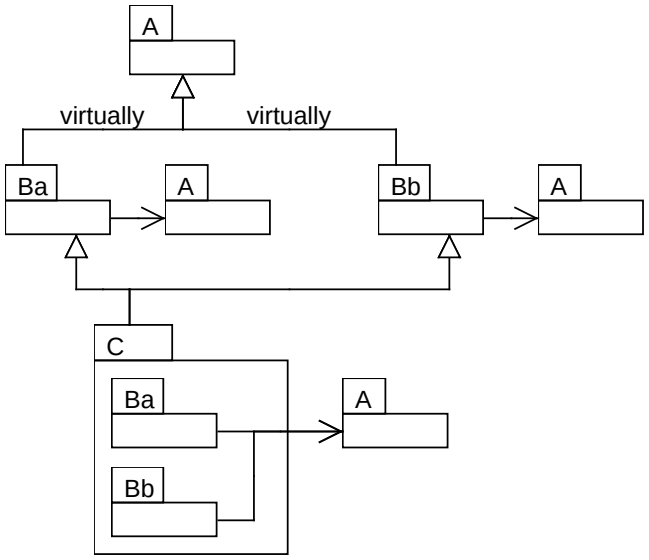


Figure 4.2: Virtual diamond. Diamond problem example with virtual inheritance.

to which declarations of built-in and user functions (and newly also methods) are converted, but constructors and destructors are special, because although they are implemented by the user, sometimes they are not called by the name of the implementation method, so they have to be called indirectly.

We decided to handle this situation by creation of special data structures in GIMPLE++, which are partially a user function and partially a built-in function. We renamed `CallableValue` (which originally was `FunctionValue`) to `UserCallableValue` and added a new base class `CallableValue`, from which we derived `UserCallableValue` and `BuiltinCallableValue`. From both those classes we derived a new class `SpecialCallableValue`, which represents the aforementioned built-in-function with possible user implementation. This inheritance is described also in Figure 4.3.

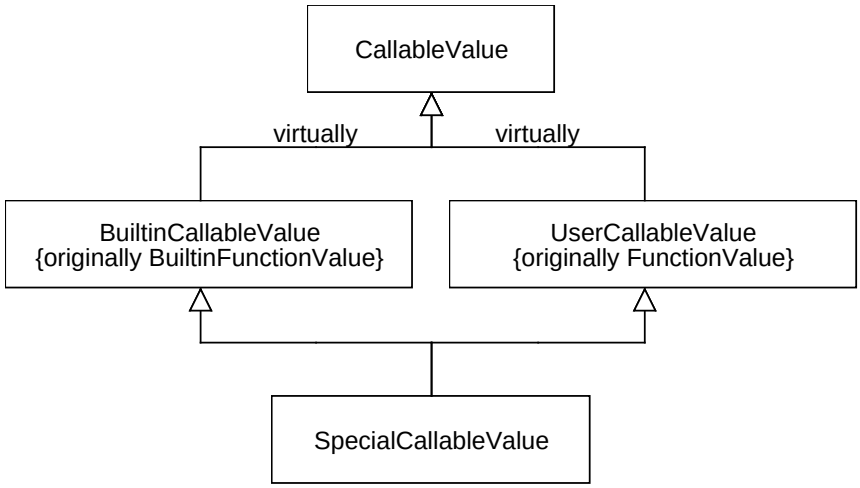


Figure 4.3: Inheritance of classes derived from CallableValue.

The main advantage of this approach in comparison with creating the class `SpecialCallableValue` on the same level of inheritance as the `UserCallableValue` and the `BuiltinCallableValue` is that the already created system of passing user implementations from the GCC to the model checker can be used almost without a change.

During the conversion from GIMPLE to GIMPLE++ also the name of the converted built-in function is converted to the identifier using a map in the variable `m_callableCodes` in the class `GimpleHelpers`. This identifier is then used to look for the implementation of this built-in function in `getBuiltinCallableMethod`. A pointer to this implementation is finally stored into the GIMPLE++ representation of the function and used by the interpreter to call the function.

The `builtinCompCtor` and the `builtinCompDtor` functions implement the functionality of the built-in constructors and destructor. Those functions look for the user method to be called according to the name, definition class and parameters.

To connect those functions with corresponding method names (which are listed at the beginning of this section) we had to create identifier entries in enumeration `callable_code`. This enumeration is also used to distinguish between built-in and user callables. So we added also a special entry, which separates the built-in callables from the special ones. This extension of course implied small modifications in the code, which uses this enumeration.

5. Pointer constants and field offsets

Every class in C++ can be derived from more than one other class (*multiple inheritance*). That causes the already mentioned diamond problem [3]. To support this inheritance with reasonable functionality, C++ allows deriving classes virtually, so they can be assembled together, when they are inherited via the diamond pattern. Because of this virtual inheritance GCC generates a lot of code, which is not directly translated from the programmer's code, and there occurred two incompatibilities in this code.

First of them is that the generated code contains integer constants erroneously typed as pointer constants although they are only offsets, so they should by typed as integer constants. GMC correctly marked this typing as erroneous (it is not a good idea to have fixed addresses in a code when dynamic allocation is used). We solved this situation by switching off this check and we added support of pointer constants into GMC. This approach has the disadvantage that GMC can no more mark this erroneous typing in the user code.

The second problem was that in GIMPLE there are defined offsets of fields in structures and classes, but GMC did not calculate with them and created a proprietary order of the fields instead. It was not a problem if only C code was processed. However, GCC generates additional code for C++ virtual inheritance; the fields' offsets differ between GCC and GMC, which results in using different places in simulated memory for a variable during interpretation. So we changed GMC to use the defined offsets. Unfortunately in GMC those offsets are calculated with byte precision, but in GIMPLE they are defined with bit precision. This difference can result in the same incompatibility as before, but in this case the situation is recognized and the error is reported. However, this situation is not common, because fields are usually aligned to beginning of bytes.

5.1 Implementation

The original code of GMC was not able to work with nonnull pointer constants, so it did not have field for the value of the pointer. We added field `m_value` into the class `PtrConstant` and into the class `PtrValue` – objects of `PtrValue` are generated from the objects of `PtrConstant`. Additionally, we had to change the `addOp` method in the class `PtrValue`, to be able to add a pointer-typed offset to a pointer.

The implementation of the second problem has been done by the implementation of proper offset support by their propagating through the whole chain from GIMPLE through `ConstructionHelpers` to the `OffsetDecl` class, where they are stored and used by the `OffsetMapper` class, which checks the byte alignment and computes the final offset.

6. Exception handling

The exception support at C++ consists of support in the language, support in the compiler and support at runtime. The support in the language consists of special constructs, which serve for throwing exceptions and marking regions for their catching. Those constructs are processed using the compiler resulting into code which is used at runtime to process exceptions.

6.1 Language support

The language support consists of the `try` block, which contains the code where exception can be thrown and the `catch` block, which contains the code which is executed after an exception of the declared type has been thrown in the `try` block. Finally, anywhere in the code there can be the `throw` command.

6.2 Compiler support

During code processing the compiler converts the `try` and `catch` blocks in the C++ code into `eh_region`s in GIMPLE. Those regions, however, are not used in GMC, because the compiler also creates edges marked as `throw`, which lead from the block where an exception can be thrown to the block where the execution should continue in this case.

```
int main(int argc, char ** argv) {
  try {
    printf("Try block\n");
  }
  catch (...) {
    printf("Catch block\n");
  }
  printf("Final block\n");
}
```

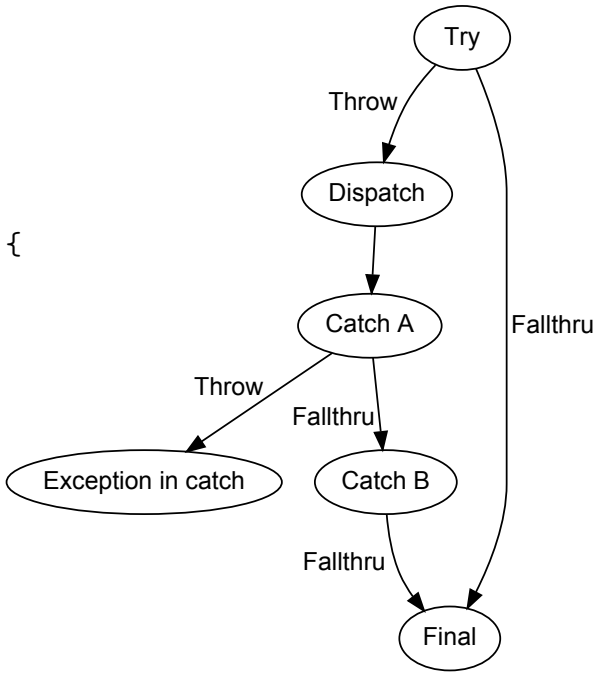


Figure 6.1: Exceptions without classes.

An example of the GIMPLE basic block diagram with code from which it is generated is in Figure 6.1. In this example you can see that if there is no exception in the Try block, the execution after finishing the Try block continues directly by the Final block. If an exception is thrown in the Try block, the execution

continues by the Dispatch block. This block is terminated by `EhDispatchStmt`, which selects the appropriate catch block (note that the edge is not marked, because there can be more edges leading to different catch blocks). A catch block is split into the blocks Catch A and Catch B. Catch A contains call of the `__cxa_begin_catch` function and the user code. Catch B contains a call of `__cxa_end_catch`. Because the exception can be thrown in the user code in catch block, there is also a special block to handle this exception.

The situation is a little bit complicated when classes are involved in the code with exceptions. Consider Figure 6.2. Two blocks (Destructor A and Destructor B) have been added because of usage of the class `A` with a user destructor. Those blocks contain almost the same code – a destructor call (in addition Destructor B contains `ResxStmt` to signal continuation by the throw edge). Destructor B is executed when an exception is thrown in `printf`. Destructor A is used during normal execution. Note that there is also a throw edge from Destructor A for the case when an exception is thrown during the destructor execution.

```
int main(int argc, char ** argv) {
  try {
    A a(21);
    printf("Try block: %d\n", a.number);
  }
  catch (...) {
    printf("Catch block\n");
  }
  printf("Final block\n");
}
```

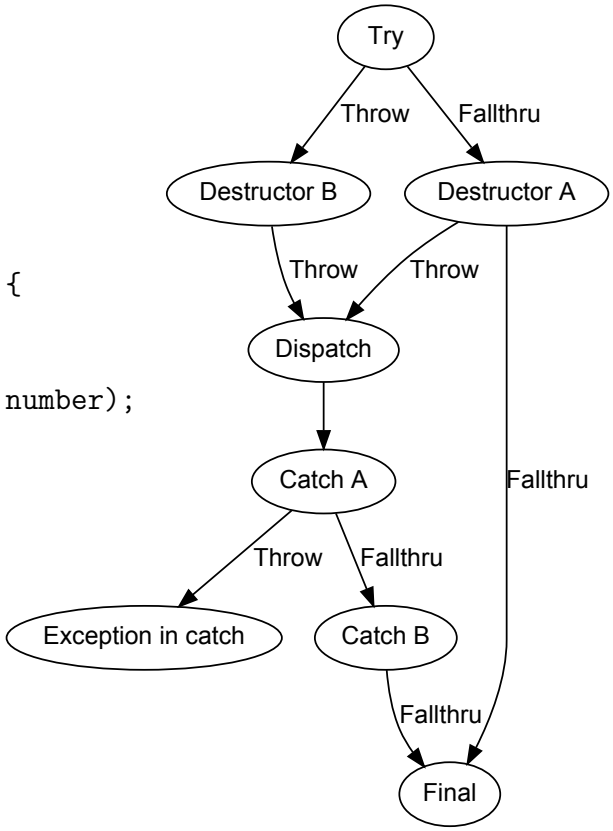


Figure 6.2: Exceptions with classes.

Until now we discussed constructs for exception catching. Now let us focus on exception throwing. When an exception is thrown using the `throw` command in C++ code, the function `__cxa_allocate_exception` is called first to allocate the memory for the exception in GIMPLE. Then a constructor is called (if there is any). And finally the exception is thrown using the `__cxa_throw` call, which starts stack unwinding. Note that there is also the `__cxa_free_exception` function, which frees the memory of the exception. This function is called from already noticed `__cxa_end_catch`.

6.3 Runtime support

Runtime support consists of implementation of mentioned built-in functions, stack unwinding and catch dispatching. All of those parts are closely related to implementation, so we precisely describe them in the following section.

6.4 Implementation

The implementation consists of built-in methods, stack unwinding and catch dispatching. Stack unwinding is used to locate and execute code after an exception is thrown. Catch dispatching is then used to find and execute the appropriate catch block. All those parts are separately described in the following subsections. During implementation of catch dispatching we faced a problem with type recognition, so also this problem is described in the subsection Catch dispatching. Additionally there is a subsection Future work, which describes what has to be done to overcome the problem.

6.4.1 Built-in functions

In the previous sections we mentioned some built-in functions, which are used during exception handling. Those functions have to be implemented also in the GMC runtime. Because of a problem, which is described later, some built-in functions are not (fully) implemented. To clarify the state of built-in function implementation, we attach Table 6.1.

original name	implementation name	implementation state
`__cxa_allocate_exception`	`builtinCxaAllocateException`	full
`__cxa_free_exception`	`builtinCxaFreeException`	full
`__cxa_throw`	`builtinCxaThrow`	partial
`__cxa_rethrow`**	`builtinCxaRethrow`	full
`__cxa_begin_catch`	`builtinCxaBeginCatch`	full
`__cxa_end_catch`	`builtinCxaEndCatch`	partial*

* Implementation of `gxxExceptionCleanup` function is missing.
** This function has not been mentioned, but it is called, when an exception is catched and then thrown again.

Table 6.1: State of implementation of exception handling builtin functions.

6.4.2 Stack unwinding

Stack unwinding, starts when an exception is thrown using `__cxa_throw` function, which continues by `ThrowStep`. This step is visited by the `Thread` class, which analyzes the block diagram using `throvv`[1] method of the `InstructionPtr` class. If there is a throw edge, it is followed. If there is no such one, the method `returnFromFunctionWithExceptions` is called. This method uses the method

[1]Method name contains two v, because `throw` is a keyword.

`returnFromFunction`, which already was in GMC, because it was used to normally return from the function.

Note that when an exception is thrown, the execution continues only by throw edges. This is not explicitly written in GMC, but when you look at `InstructionPtr::stepOverBlocks` method, you will see that if there is an exception thrown, only throw edge is used. Additionally according to empiric experiences, every basic block, which is executed after the exception has been thrown (usually it contains destructor call), finishes by the `ResxStmt`.

This whole process can finish by returning from the `main` method or the execution can arrive to the `EhDispatchStep`.

6.4.3 Catch dispatching

Catch dispatching is a process of selection the appropriate catch block for the thrown exception. In the GIMPLE++ it is symbolized by `EhDispatchStep`, which is located in the last basic block before the catch block. And here is a gap in our implementation of the exception handling. The problem is in type comparison. When an exception is thrown, the type of the exception is available in two formats. The first one of those formats is the structure `type_info`, which is passed to the function `__cxa_throw` as a parameter. The second format is GIMPLE++ type format, because we can, at runtime, get the type of the value which is thrown.

On the other hand the types in list for a catch block are in an unknown format, which could possibly be convertible to the `type_info` format, but there is no information about the type of the base class, which is needed to validate if the thrown type is a subtype of a type in the list of the catch block. This validation is necessary to decide, if the catch block catches the exception or not. So we need to pass the information about type inheritance into the runtime from the front-end of the compiler. This could be done in a variety of ways and because there is also a related problem – dynamic type casting, which is not implemented yet, we decided to do more research before finishing the implementation of exception handling.

6.4.4 Future work

Now the code is prepared for creating GIMPLE++ structure, which can hold information about types in catch blocks. This code is at the end of the `gb_process_cfun` function and it is commented out. The mentioned structure is designed to hold types in GIMPLE++ format, but it could be rewritten to other type format when needed. There is also a partially implemented selection of a catch block: in `GimpleStmtInterpreter` there is the `visit` method for the `EhDispatchStmt` statement, in which getting of the thrown exception type is missing. In this method the `findCatchForType` method is also called. That function has to be implemented to find the appropriate catch block for the passed type of exception. During this process the information about inheritance has to be used, but there is no prepared data structure to do that. After those methods and built-in functions from Table 6.1 will be fully implemented, the exception

handling system should be prepared for testing, which is necessary, because it has not been tested much yet.

7. Templates

Templates allow the programmer to create classes with the same functionality for more than one type at once. When we considered support of C++, we supposed that templates are translated into non-templated GIMPLE before the GMC processes them. This assumption has been approved, when we tried to interpret code with a templated class, which has been specialized for simple types such as `int`, `float` and `char`. But when we specialized this class for complex type (`struct` or `class`) and tried to access a variable of the templated type, GMC stopped working. We did not fully investigate the situation, because template support is not a subject of this work, but we realised that there is a problem with the field declaration, which is missing. We also realised, that this problem is sometimes not present (probably when GMC is runned on 64-bit machine).

8. Other modifications

There are also other small changes, which we did in GMC. We shortly describe them in this chapter.

8.1 Enumerations

Enumerations are represented by integer constants in GIMPLE generated from the C code and they are represented by enumeration constants in GIMPLE generated from the C++ code. So we adapted the implementation of following representation of enumerations. Enumeration constants are represented by the `EnumeralValue` class and marked by `EnumDeclType` or `EnumConstType`, which are both derived from `EnumType`.

8.2 Basic block diagram export

Before implementation of exception handling we needed to analyze the basic block diagram in GIMPLE. To do this we created the function `dumpCallable` in the class `ConstructionHelpers`, which exports the basic block diagram into a format, which is readable by the program Graphviz, which is able to create a picture from this format. Calling this function is commented out at the end of the method `gb_process_cfun`.

8.3 References

A reference is also a construct, which is available only in C++, so we had to implement support for it into GMC. We created the classes `RefType` and `RefValue`, whose implementation is inspired by the classes `PtrType` and `PtrValue`.

8.4 Saving through pointer

During testing references we accidentally used saving into a substructure through a pointer. Unfortunately this saving did not work. After analysis we realized that problem was in creating `RecordValue` with zero offset in `ValueReader`. So we fixed it by setting correct offset in the constructor.

9. Evaluation

The main aim of this work was to implement into GMC the support of the basic C++ features: inheritance, constructors, destructors, virtual methods and exception handling. Most of those features have really been implemented, but exception handling has been implemented only partially because of its complexity and relation to dynamic type casting. Additionally we also analyzed support of templates.

9.1 Tests

In this section we describe the tests in Appendix B to simplify their understanding and to clarify their coverage.

constructors.cpp This test is aimed to test implementation of ordinary constructors and destructors. Different numeric types are used to test how constructor is called with an implicit conversion of a parameter. Note that format "%.0f" is used to overcome an error in the implementation of the `printf` function in GMC, which causes a wrong output of floats in the default formatting.

derived_constructors.cpp This test verifies the implementation of switching between base and complete constructors. For this, the virtual inheritance is needed, so it is tested by this test also (by diamond inheritance).

enum.cpp, enum2.cpp The test `enum.cpp` is a basic test of the C++ enums implementation. The test `enum2.cpp` focuses on verifying numeric operations on enumeration values. Those tests were derived from the original tests of the C implementation in GMC.

method_overload.cpp This test is focused on testing overloadability of methods. This functionality is handled by GCC before the GMC is involved, so this was not a part of our implementation.

templates.cpp Templated class is declared in this test and then it is instantiated using `int`, `float` and `SimpleClass`. All those instantions work, but when the variable inside `SimpleClass` is used, GMC crashes, because the offset of this variable is not found. You can find more precise description of this problem in chapter 7 Templates.

templates2.cpp The same test as templates.cpp, but here constructors are not used to allow testing templates when there is an error in constructor implementation.

virtual_methods.cpp This test verifies if GMC correctly interprets the virtual methods, their overriding and calling using a casted pointer.

global_test.cpp This test verifies the usage of classes together with threads. There is the `Thread_arg` class, from which the `Thread_value_arg` is derived. Those classes are used as parameter of the `thread_function` function, which is called in new threads from the `main` function. There is also a race condition – variable `arg1` is initialized after starting the thread, so it is not deterministic, which value will the thread use.

9.1.1 Failing tests

If you try to run tests, you will realize that some of them fail. Some tests are designed to test unimplemented features to emphasize this fact. GCC sometimes generates other numbers for temporary variables too, which also causes failing tests. We provide a list of tests, which normally fail.

c/basictests/integers01.c Inherited from previous version of GMC.

c/basictests/mm_unreachable.c Inconsistence in variable numbering.

c/basictests/rand_mc_2.c Inconsistence in variable numbering.

c/basictests/trialall.c Uses unsupported bit-precision offsets (see chapter 5 Pointer constants and field offsets).

c/basictests/void2.c Inherited from previous version of GMC.

c/basictests/void4.c Inherited from previous version of GMC.

c/basictests/void5.c Inherited from previous version of GMC.

c/functiontests/assert.c Inconsistence in variable numbering.

c/functiontests/mutex1.c Inherited from previous version of GMC.

c/functiontests/strcat.c Inherited from previous version of GMC.

c/functiontests/thread1.c Inherited from previous version of GMC.

cpp/basictests/templates.c Missing variable offset (see chapter 7 Templates).

cpp/basictests/templates2.c Missing variable offset (see chapter 7 Templates).

When another test fails, the user should check the `.test_result` and `.test_result_err` files first to find the reason of the failure. The listed tests are only from directories, from which the `run_tests.sh` script runs the tests.

10. Related work

Except for GMC, there exist other verification tools for C++ code. In this chapter we describe some of them and we compare them with GMC.

10.1 Types of model-checking

Different types of model-checking are used in related tools, so we explain the basic concepts here.

Explicit state model-checking is a method which is used in GMC and it exactly simulates the program step by step and when it arrives to the end of program, it backtracks and tries another branch.

Symbolic model-checking does not instantiate all program states, but considers them in groups, which has similar properties. Graphs which describe possible transitions between those groups are then used to verify the program. Usually an SMT solver is used for this purpose.

Bounded model-checking is special case of symbolic model checking. In addition it limits the number of states; the traces are cut by various methods after n steps, e. g. number of loop iterations is limited.

10.2 MCP

Probably the most important C++ code explicit state model checker is MCP from NASA. Their web page describes MCP like this:

> *"MCP is an explicit-state software model checker that supports the entire C++ programming language."*
>
> *"The MCP (Model checker for C Plus plus) model checker was constructed specifically to allow programs written in C or C++ to be model-checked directly without requiring prior translation or model extraction. It builds upon the LLVM compiler infrastructure, thereby avoiding the requirement to directly recognise the (extremely complex) C++ language at source-level. This approach has allowed us to support the entire C++ language, including templates. The C language is handled fully, not just as an improper subset of C++."* [4]

The work on MCP started approximately at the same time as the work on GMC. It also works on an intermediate language, but it uses LLVM, which is intended for this purpose and is better described. The basic architecture of MCP is very similar to the architecture of GMC, but the functionality set is wider in MCP thanks to a bigger developer team, so it fully supports templates for example.

10.3 StEAM

> *"StEAM is a model checker for C++. It detects deadlocks, segmentation faults, out of range variables and non-terminating loops."* [5]

This tool uses the same principles, but it uses the machine code from the *Internet C++ Virtual Machine*. Except for capabilities enumerated in the citation, it also supports multithreaded applications. There is also the StEAM-XXL version, which can check bigger programs thanks to saving data to the hard disk. StEAM is available only for Linux.

10.4 LLBMC

> *"LLBMC (the low-level bounded model checker) is a static software analysis tool for finding bugs in C (and, to some extent, in C++) programs. It is mainly intended for checking low-level system code and is based on the technique of Bounded Model Checking."* [6]

Also this model checker uses the intermediate language from LLVM, but uses bounded checking, so the number of steps in one path of the program is limited. Model checking is done by an SMT solver in this tool.

10.5 ESBMC++

> *"ESBMC is a context-bounded model checker for embedded C/C++ software based on Satisfiability Modulo Theories (SMT) solver. It allows the verification engineer (i) to verify single- and multi-threaded software (with shared variables and locks); (ii) to reason about arithmetic under- and overflow, pointer safety, memory leaks, array bounds, atomicity and order violations, deadlock and data race; (iii) to verify programs that make use of bit-level, pointers, structs, unions and fixed-point arithmetic."* [7]

This model checker uses the goto-cc compiler to create a GOTO-program from the C/C++ code. This program is then analysed using an SMT solver. So this model checker uses quite different approach than GMC.

10.6 CBMC

> *"CBMC is a Bounded Model Checker for ANSI-C and C++ programs. It also supports SystemC using Scoot. It allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. Furthermore, it can check ANSI-C and C++ for consistency with other languages, such as Verilog."* [8]

CBMC uses the same approach as ESBMC++, but it does not use an external compiler of C/C++. It compiles the sources on its own into a control flow graph. Then it unwinds the loops and checks the resulting program using an SMT solver. In addition this tool allows the user to check consistency of the code with the code in other language (such as Verilog). This tool is available for Linux, Windows and Mac.

10.7 Feature summary

In Table 10.1 we present a summary of features of discussed model checkers including GMC for easy comparison of their abilities. The first group of the rows contains language features and the second contains program properties, which checkers are able to verify. Unfortunately we were unable to find all information. Some cells are thus empty. In the case of MCP the official web page said that it fully supports the C++ language. This information is presented in our table by ticking all the language features for this checker.

feature	GMC	MCP	StEAM	LLBMC	ESBMC++	CBMC
floating-point values	✓	✓		✗		
bit fields	✗	✓		✓	✓	
multithreading	✓	✓	✓		✓	
exceptions	✗	✓				✓
templates	✗	✓				
deadlocks	✓	✓	✓		✓	
illegal memory access	✓	✓	✓	✓	✓	✓
non-terminating loops	✓		✓			
uninitialized variables	✓		✓			
integer overflow	✗			✓	✓	
division by zero	✓			✓		
invalid memory free	✓			✓		

Table 10.1: Summary table of feature support.

For creation of this table official web pages of tools already referenced from citations and web page of the TACAS 2013 competition [9] have been used.

11. Conclusion

Our aim to improve GMC to support basic features of C++ has been successfully accomplished, so GMC now supports inheritance, virtual inheritance, constructors, destructors and virtual methods. The only part which has not been finished is the support for exceptions. GMC is prepared to implement other features of C++.

As future work, first thing, which should be done, is finishing the implementation of exceptions. Except that the main feature, which is missing in the C++ implementation is the C++ library. Also precision of offset calculation should be improved (more precise description of this problem is in chapter 5 Pointer constants and field offsets). Another useful feature would be full support for templates (more information is in chapter 7 Templates). And there are also some internal problems, which should be solved – they are marked in the code (e. g. inconsistent usage of the method `isSame`).

Bibliography

[1] Jan Kouba. *Memory Representation for Model Checker of C/C++*. Praha: MFF UK 2009. Diplomová práce, MFF UK, Katedra softwarového inženýrství.

[2] Ondřej Krč-Jediný. *GIMPLE Model Checker*. Praha: MFF UK 2010. Diplomová práce, MFF UK, Katedra distribuovaných a spolehlivých systémů.

[3] *Multiple inheritance*. [online]. Wikipedie [cit. 2013-04-13]. Available from: `http://en.wikipedia.org/w/index.php?title=Multiple_inheritance&oldid=546468022#The_diamond_problem`

[4] *Analysis Tools for C++*. [online]. NASA [cit. 2013-04-21]. Available from: `http://ti.arc.nasa.gov/tech/rse/vandv/analysistoolsc/`

[5] *StEAM*. [online]. University of Dortmund [cit. 2013-04-21]. Available from: `http://steam.cs.uni-dortmund.de/`

[6] *LLBMC*. [online]. Karlsruhe Institute of Technology [cit. 2013-04-23]. Available from: `http://llbmc.org/`

[7] *ESBMC*. [online]. University of Southampton, School of Electronics and Computer Science [cit. 2013-05-12]. Available from: `http://esbmc.org/`

[8] *CBMC*. [online]. University of Oxford, Systems Verification Group [cit. 2013-05-12]. Available from: `http://www.cprover.org/cbmc/`

[9] *TACAS 2013*. [online]. European joint conferences on theory & practice of software [cit. 2013-06-22]. Available from: `http://sv-comp.sosy-lab.org/2013/results/index.php`

[10] *GNU Compiler Collection (GCC) Internals*. [online]. GCC team [cit. 2013-07-27]. Available from: `http://gcc.gnu.org/onlinedocs/gccint/`

Appendices

Appendix A: User guide

If you want to run GMC, you will have to go through the following list of steps. The list for B variant is also in the file `sources/INSTALL` on the attached DVD. If you get in trouble during installation, continue reading, there are some important notes later in the text.

0. Before you can compile and run GMC, you have to install following packages. Only commonly not installed packages are listed. If you will use the already compiled binaries of GMC for 32-bit Ubuntu (according to variant A of this guide), you will probably not need all of those packages.

   - A POSIX or SVR4 awk
   - GNU binutils (required only in some circumstances)
   - gzip version 1.2.4 (or later) or bzip2 version 1.0.2 (or later)
   - GNU make version 3.80 (or later)
   - GNU tar version 1.14 (required only on some platforms)
   - GNU Multiple Precision Library (GMP) version 4.3.2 (or later)
   - MPFR Library version 2.4.2 (or later)
   - MPC Library version 0.8.1 (or later)
   - libelf version 0.8.12 (or later)
   - boost C++ libriaries version 1.45.0 (or later to 1.46.1) – GMC does not work with boost in version 1.52.0

   If you want to use already compiled GMC binaries for 32-bit Ubuntu, continue by the step A. If you want to compile them on your own, continue by the step B.

A. Use binaries (compiled for 32-bit Ubuntu) from the attached DVD.

   1. Copy the binaries of GMC from the attached DVD to your hard drive.

      ```
      > mkdir gmc
      > cp -R /media/dvd/program/* gmc
      > cd gmc
      ```

   2. Allow reading, writing and executing in the GMC directory.

      ```
      > chmod -R u+rwx .
      ```

   3. Set environment variables.

      ```
      > . ./setEnv.sh
      ```

   Now GMC should be prepared for execution.

B. Compile binaries from the sources on the attached DVD.

   1. Copy the sources of GMC from the attached DVD to your hard drive.

```
> mkdir gmc
> cp -R /media/dvd/sources/* gmc
> cd gmc
```

2. Set environment variables.

   ```
   > . ./setEnv.sh
   ```

3. Compile GMC.

   ```
   > make
   ```

4. Download, patch and compile GCC

   ```
   > ./prepare_gcc.sh
   ```

Now GMC should be prepared for execution.

If all those steps are finished successfully, you can use the GMC. So you can try to run tests:

```
> cd tests
> ./run_tests.sh
> cd ..
```

Or you can check your own code (you can find more information about running GMC with your own code in the file `sources/dist/README` on the attached DVD):

```
> cd dist
> ./GMC g++ -m your_cpp_file.cpp #Executes model-checking
> cat ModelChecker_stderr #Shows results of model-checking
> cd ..
```

In case of error When you receive some error message, try to go through this list, where are described some known issues.

- If you received error about missing file `ansidecl.h` during execution of `make` command, use the following command to use backup file from the attached DVD and try to run the command `make` again.

  ```
  > cp ./ansidecl.h.backup ./mc/ansidecl.h
  ```

- If another file is missing, make sure that you executes the command from the right working directory. All pieces of code in this guide should be executed from the `gmc` directory created in the step 1 (except of those pieces of code in steps 1, of course).

- If the compilation of the file `mc/git/Serializer.cpp` fails, it could be caused by insufficiency of memory (try to terminate some applications). The compilation of the mentioned file is usually much longer than other files (5 minutes was normal time on our machine), but when compilation takes much longer, it is sign of the memory insufficiency, which finally results in some error.

- If you did not succeeded with some step before or in variant B, try to look in file `sources/INSTALL` on the attached DVD, there are some additional information about the installation.
- If you try to run commands from the file `prepare_gcc.sh` manually and you change the GMC sources, you will have to run following command from the `gmc` directory.

  ```
  > touch gcc-4.5.0/gcc/lto/gimplexx-bridge.c
  ```

  It is because the library for GCC has to be recompiled in case of change of shared sources and those shared sources are not mentioned in the GCC makefiles (to simplify the development), only `gimplexx-bridge.c` is.

Appendix B: Tests

constructors.cpp

```
1  #include <cstdio>
2
3  class A {
4    private:
5      int a;
6      double b;
7    public:
8      A(int a) :a(a), b(0) {
9        printf("Running constructor A(int %d).\n", a);
10     }
11
12     A(double b) :a(0), b(b) {
13       printf("Running constructor A(double %.0f).\n", b);
14     }
15
16     A(int a, double b) {
17       printf("Running constructor A(int %d, double %.0f).\n", a, b);
18       this->a = a;
19       this->b = b;
20     }
21
22     ~A() {
23       printf("Running destructor ~A().\n");
24     }
25
26     void printA() {
27       printf("a=%d, b=%.0f", a, b);
28     }
29
30     void printAn() {
31       printA();
32       printf("\n");
33     }
34 };
35
36 class B : public A {
37   private:
38     int cc, *c;
39   public:
40     B(int a) :A(a), cc(42) {
41       printf("Running constructor B(int %d).\n", a);
42       c = &cc;
43     }
44
45     B(double b) :A(b), cc(42) {
46       printf("Running constructor B(double %.0f).\n", b);
47       c = &cc;
48     }
49
50     B(int a, int b, int *c) :A(a, b), cc(42) {
51       printf("Running constructor B(int %d, int %d, int %d).\n",
52         a, b, *c);
53       this->c = c;
54     }
55
56     ~B() {
57       printf("Running destructor ~B().\n");
58     }
59
60     void printB() {
61       printA();
62       printf(", *c=%d, cc=%d", *c, cc);
63     }
64
65     void printBn() {
```

```
66        printB();
67        printf("\n");
68      }
69  };
70
71  int main(int argc, char ** argv) {
72    int n = 21;
73
74    A a1(1), a2(2.0), a3(3, 3.0), a4(4, 4), a5((int)5.0, 5);
75    B b1(1), b2(2.0), b3(3, (int)3.0, &n), b4(4, 4, &n);
76
77    a1.printAn();
78    a2.printAn();
79    a3.printAn();
80    a4.printAn();
81    a5.printAn();
82
83    b1.printAn();
84    b1.printBn();
85    b2.printAn();
86    b2.printBn();
87    b3.printAn();
88    b3.printBn();
89    b4.printAn();
90    b4.printBn();
91  }
```

derived_constctors.cpp

```
1   #include<cstdio>
2
3   class A {
4     public:
5       int id;
6       A(int id):id(id) {
7         printf("Running constructor A(int %d).\n", id);
8       }
9
10      ~A() {
11        printf("Running destructor ~A().\n");
12      }
13  };
14
15  class Aa : public virtual A {
16    public:
17      int number;
18      Aa():A(0), number(21) {
19        printf("Running constructor Aa().\n");
20      }
21
22      ~Aa() {
23        printf("Running destructor ~Aa().\n");
24      }
25
26      virtual void echo() {
27        printf("Aa::echo(): Aa::number=%d\n", number);
28      }
29  };
30
31  class Ab : public virtual A {
32    public:
33      int number;
34      Ab():A(1), number(42) {
35        printf("Running constructor Ab().\n");
36      }
37
38      ~Ab() {
39        printf("Running destructor ~Ab().\n");
40      }
41
```

```
42      virtual void echo() {
43        printf("Ab::echo(): Ab::number=%d\n", number);
44      }
45  };
46
47  class B : public Aa, public Ab {
48    public:
49      B():A(2) {
50        printf("Running constructor B().\n");
51      }
52
53      B(int n):A(3) {
54        printf("Running constructor B(int %d).\n", n);
55        Aa::number = n;
56        Ab::number = 2 * n;
57      }
58
59      ~B() {
60        printf("Running destructor ~B().\n");
61      }
62
63      void echo() {
64        printf("B::echo(): A::id=%d Aa::number=%d Ab::number=%d\n",
65          A::id, Aa::number, Ab::number);
66      }
67  };
68
69  class C : public B {
70    public:
71      C():A(4) {
72        printf("Running constructor C().\n");
73      }
74
75      ~C() {
76        printf("Running destructor ~C().\n");
77      }
78  };
79
80  int main(int argc, char** argv) {
81    A a(-1);
82    Aa aa;
83    aa.echo();
84    Ab ab;
85    ab.echo();
86    B b;
87    b.echo();
88    ((Aa)b).echo();
89    ((Ab)b).echo();
90    C c;
91    c.echo();
92  }
```

enum.cpp

```
1   #include <stdio.h>
2
3   int main(int argc, char** argv)
4   {
5     enum Days {monday, tuesday, wednesday,
6       thursday, friday, saturday, sunday};
7
8     enum Days TheDay;
9     int j = 2;
10
11    TheDay = (enum Days)j;
12
13    if (TheDay == saturday || TheDay == sunday)
14      printf("Hurray it is the weekend\n");
15    else
16      printf("Curses still at work\n");
```

```
17
18    return (0);
19  }
```

enum2.cpp

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5     enum DAY {          /* Defines an enumeration type     */
6         saturday,       /* Names day and declares a        */
7         sunday = 0,     /* variable named workday with     */
8         monday,         /* that type                       */
9         tuesday,
10        wednesday,      /* wednesday is associated with 3 */
11        thursday,
12        friday
13    } workday;
14
15    enum DAY today = wednesday;
16    enum DAY tomorrow = ( enum DAY ) ( today + 1 );
17    enum DAY beforeyesterday = ( enum DAY ) ( today - 2 );
18    enum DAY day1 = ( enum DAY ) ( 2 - 1 );
19    enum DAY day1a = ( enum DAY ) ( today / tuesday );
20    enum DAY day6 = ( enum DAY ) ( today * 2 );
21    enum DAY day0 = ( enum DAY ) ( today % wednesday );
22
23    if (wednesday == 3)
24      printf("Comparison OK\n");
25
26    printf("3=%d, 4=%d, 1=%d, 1=%d, 1=%d, 6=%d, 0=%d\n",
27      today, tomorrow, beforeyesterday, day1, day1a, day6, day0);
28
29    enum BOOLEAN  // Declares an enumeration data type called BOOLEAN
30    {
31        FALSE,      // false = 0, true = 1
32        TRUE
33    };
34
35    enum BOOLEAN end_flag, match_flag = (enum BOOLEAN)0;
36      // Two variables of type BOOLEAN
37
38    if ( match_flag == FALSE )
39    {
40      printf("FALSE=%s\n", "FALSE");
41    }
42    end_flag = TRUE;
43
44    return 0;
45  }
```

method_overload.cpp

```
1   #include<cstdio>
2
3   class A {
4     public:
5       int number;
6
7       void echo() {
8         printf("A.echo(): %d\n", number);
9       }
10
11      void echo(int i) {
12        printf("A.echo(%d): %d\n", i, number);
13      }
```

```
14 };
15
16 int main(int argc, char** argv) {
17   A a;
18   a.number = 21;
19   a.echo();
20   a.echo(42);
21   printf("Finished.\n");
22 }
```

templates.cpp

```
1  #include <cstdio>
2
3  class SimpleClass {
4    public:
5      int number;
6  };
7
8  template <class T>
9  class TemplatedClass {
10   public:
11     T value;
12
13     TemplatedClass(T value):value(value) {  }
14
15     T get() { return value; }
16 };
17
18 int main(int argc, char** argv) {
19   SimpleClass c;
20   c.number = 42;
21   TemplatedClass<int> tint(2);
22   TemplatedClass<float> tfloat(4.2);
23   TemplatedClass<SimpleClass> tclass(c);
24   printf("int: %d\n", tint.get());
25   printf("float: %f\n", tfloat.get());
26   printf("SimpleClass: %d\n", tclass.get().number);
27   printf("Finished.\n");
28 }
```

templates2.cpp

```
1  #include <cstdio>
2
3  class SimpleClass {
4    public:
5      int number;
6  };
7
8  template <class T>
9  class TemplatedClass {
10   public:
11     T value;
12
13     //TemplatedClass(T value):value(value) {  }
14     void setValue(T value) { this->value = value; }
15
16     T get() { return value; }
17 };
18
19 int main(int argc, char** argv) {
20   SimpleClass c;
21   c.number = 42;
22   TemplatedClass<int> tint; tint.setValue(2);
23   TemplatedClass<float> tfloat; tfloat.setValue(4.2);
24   TemplatedClass<SimpleClass> tclass; tclass.setValue(c);
```

```
25    printf("int: %d\n", tint.get());
26    printf("float: %f\n", tfloat.get());
27    printf("SimpleClass: %d\n", tclass.get().number);
28    printf("Finished.\n");
29  }
```

virtual_methods.cpp

```
1   #include<cstdio>
2
3   class A0 {
4     public:
5       virtual void echo() {
6         printf("A0.echo()\n");
7       }
8   };
9
10  class A {
11    public:
12      int number;
13
14      A():number(21) {
15        printf("A(): number:=%d\n", number);
16      }
17
18      virtual void echo() {
19        printf("A.echo(): number=%d\n", number);
20      }
21  };
22
23  class B : public A {
24    public:
25      B() {
26        number = 42;
27        printf("B(): number:=%d\n", number);
28      }
29
30      void echo() {
31        printf("B.echo(): number=%d\n", number);
32      }
33  };
34
35  int main(int argc, char** argv) {
36    A0 a0;
37    A a;
38    B b;
39    A* c;
40    a0.echo();
41    a.echo();
42    b.echo();
43    c = &b;
44    c->echo();
45    printf("Finished.\n");
46  }
```

global_test.cpp

```
1   #include <cstdlib>
2   #include <cstdio>
3   #include <cstring>
4   #include <pthread.h>
5   #include <assert.h>
6
7   class Thread_arg {
8     private:
9       int id;
10    public:
```

```
11      Thread_arg(int const id)
12        :id(id)  {}
13
14      int get_id() const {
15        return id;
16      }
17
18      void virtual echo() const {
19        printf("Thread_arg:\n\tid=%d\n", id);
20      }
21  };
22
23  class Thread_value_arg: public Thread_arg {
24    private:
25      char * value;
26    public:
27      Thread_value_arg(int const id, const char * const value)
28        :Thread_arg(id) {
29        this->value = NULL;
30        set_value(value);
31      }
32
33      const char * get_value() const {
34        return value;
35      }
36
37      void set_value(const char * const value) {
38        if (this->value != NULL)
39          free(this->value);
40        if (value == NULL) {
41          this->value = NULL;
42        }
43        else {
44          int length = strlen(value);
45          this->value = (char *)malloc(length + 1);
46          strncpy(this->value, value, length + 1);
47        }
48      }
49
50      void virtual echo() const {
51        Thread_arg::echo();
52        printf("\tThread_value_arg:\n\t\tvalue=%s\n", value);
53      }
54  };
55
56  void * thread_function(void * arg) {
57    Thread_arg * thread_arg = static_cast<Thread_arg *>(arg);
58    Thread_value_arg * thread_value_arg = NULL;
59    if (thread_arg->get_id() != 0) {
60      thread_value_arg = static_cast<Thread_value_arg *>(thread_arg);
61    }
62    if (thread_value_arg == NULL) {
63      printf("Thread %d: No value in args.\n", thread_arg->get_id());
64      return NULL;
65    }
66    else {
67      if (thread_value_arg->get_value() == NULL) {
68        printf("Thread %d: NULL\n", thread_value_arg->get_id());
69        return NULL;
70      }
71      else {
72        printf("Thread %d: '%s'\n",
73          thread_value_arg->get_id(), thread_value_arg->get_value());
74        assert(
75          strcmp("original value", thread_value_arg->get_value()) != 0);
76        return NULL;
77      }
78    }
79  }
80
81  int main(int argc, char ** argv) {
82    pthread_t t0;
83    pthread_t t1;
```

```
84    pthread_t t2;
85
86    Thread_arg arg0(0);
87    printf("Creating thread 0\n");
88    pthread_create(&t0, NULL, thread_function, &arg0);
89
90    Thread_value_arg arg1(1, "original value");
91    printf("Creating thread 1\n");
92    pthread_create(&t1, NULL, thread_function, &arg1);
93    arg1.set_value("new value");
94
95    Thread_value_arg arg2(2, NULL);
96    printf("Creating thread 2\n");
97    pthread_create(&t2, NULL, thread_function, &arg2);
98
99    printf("Joining threads\n");
100   pthread_join(t0, NULL);
101   pthread_join(t1, NULL);
102   pthread_join(t2, NULL);
103 }
```

Appendix C: DVD content

Here we attach commented list of directories which are situated on the attached DVD.

backup Contains copy of all other directories to backup them.

document Contains this document in PDF.

program Contains GMC binaries compiled for 32-bit Ubuntu (copy it to writable media before running it).

programmer_documentation Contains programmer documentation of GMC generated from comments in the code (root file is `index.html`).

sources Contains sources of GMC.